Finite State Machines and Regular Expressions

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

- Be able to explain the problem of parsing.
- Know how to recognize a word using an NFA or a DFA.
- Know the difference between a DFA and an NFA
- Be able to convert a NFA into a DFA
- Vocabulary to know: deterministic, nondeterministic, lexing, scanning, accept state, transition.
- Know the syntax of regular expressions.
- Know how to convert between regular expressions and state machines.
- Know the limitations of regular languages.

For further reading see the Dragon Book, §3.1 and 3.6.

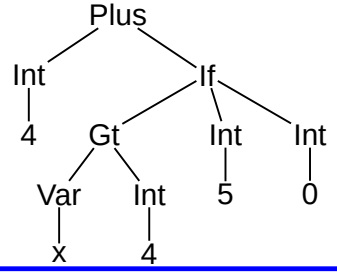
- Computer programs are entered as a stream of ASCII (usually) characters.

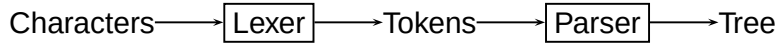
```
4 + if x > 4 then 5 else 0
```

- We want to convert them into an *Abstract Syntax Tree*

ML code

```
1 PlusExp(
2   IntExp 4,
3   IfExp(
4     GtExp(VarExp "x",
5           IntExp 4),
6     IntExp 5,
7     IntExp 0))
```





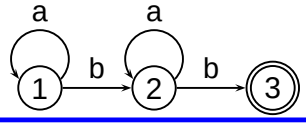
The conversion from strings to trees is accomplished in two steps.

- First, convert the stream of characters into a stream of *tokens*.
  - This is called *lexing* or *scanning*.
  - Turns characters into words and categorizes them.
- Second, convert the stream of tokens into an abstract syntax tree.
  - This is called *parsing*.
  - Turns words into *sentences*.

DFAs

Suppose I want to teach the computer how to recognize a word with the following properties:

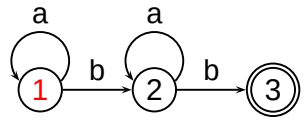
- It consists only of the letters a and b.
- The letter b occurs twice, once at the very end.
- We can use a *state machine*...



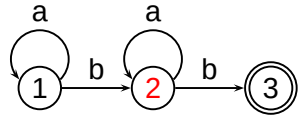
- The circles represent *states*. Start at the first one.
- The arrows represent *transitions*.
- Consume a character from the input. If it matches a transition, follow it to the next state. (Otherwise, fail.)
- If you are in the double-circle state when the input runs out, succeed.

Example: parse aabab

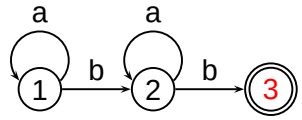
- Start at state 1:



- Read the first two letters a, and stay at state 1.
- Read b, and go to state 2:



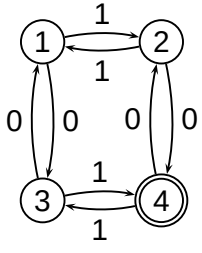
- Read an a and stay at state 2.
- Read b, and go to state 3:



- Input is over, and we're in the final state. Accept the input.

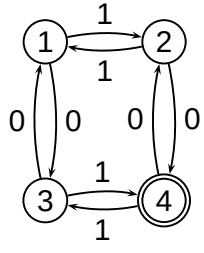
Example

What kinds of strings will this state machine accept?



Example

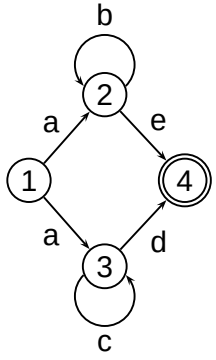
What kinds of strings will this state machine accept?



Answer: Strings of 0s and 1s with an odd number of 0s and an odd number of 1s.

State machines can be *nondeterministic* in two ways:

- Way 1: Multiple edges from a state with the same label

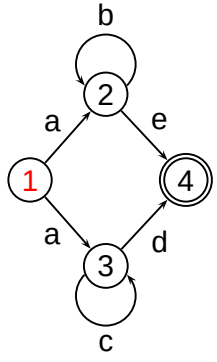


Accepted strings:

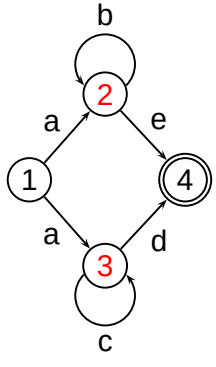
- a, any number of b, e
- a, any number of c, d

When the first a is read, the machine is in *either* state 2 or 3.

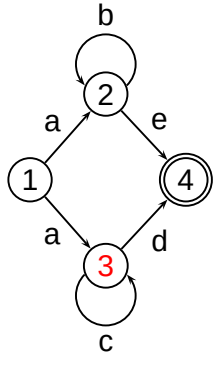
Only on the next input do you find out for sure which state is active.



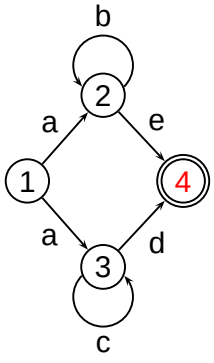
- Start at state 1



- Start at state 1
- Read an a — go to states 2 and 3



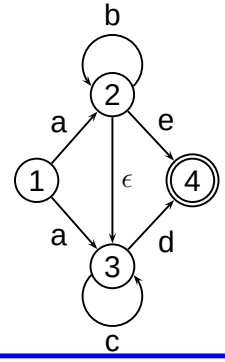
- Start at state 1
- Read an a — go to states 2 and 3
- Read a c — go to state 3



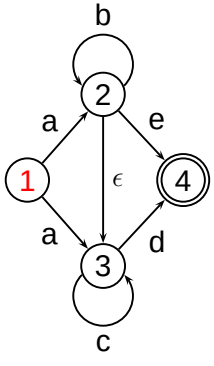
- Start at state 1
- Read an a — go to states 2 and 3
- Read a c — go to state 3
- Read a d — go to state 4 Input is over, so accept.

Way 2: Epsilon (ϵ) transitions

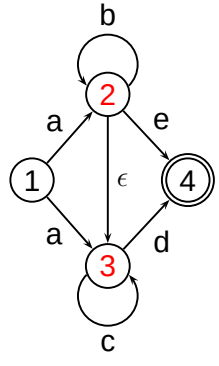
- An ϵ-transition says we can go from one state to another without any input.



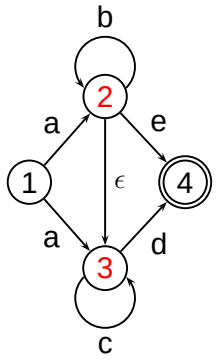
When we are in state 2, we can go to state 3 at any time, just because we feel like it.
How does this change what strings we can read?



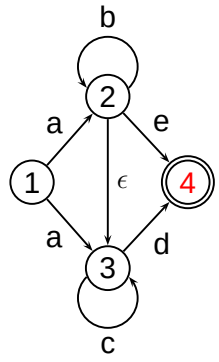
- Start at state 1



- Start at state 1
- Read an a — Go to state 2 and 3



- Start at state 1
- Read an a — Go to state 2 and 3
- Read a b — Go to state 2 and 3

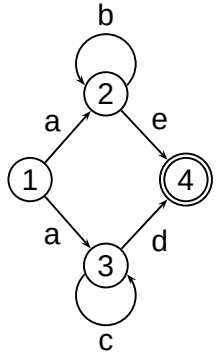


- Start at state 1
- Read an a — Go to state 2 and 3
- Read a b — Go to state 2 and 3
- Read a d — Go to state 4
  Since input is over, we accept.

- With the exception of Prolog, computers have a hard time dealing with nondeterministic state machines.
- Solution: we can convert them!

How to do it:

1. Add set {1} to the queue.
2. Pop set of states L from the queue. If seen before, discard and go to 1.
3. Take the *epsilon closure* of L to get $\hat{L}$.
4. Create a new set of states for each input recognized, push them onto the queue.

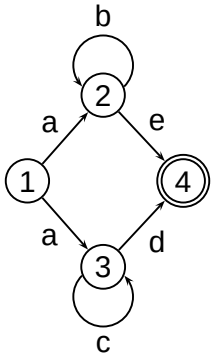


Start with a new state $P = \{1\}$. Inputs on P go to states 2 and 3.

State	Set	a	b	c	d	e
P	1	2,3				

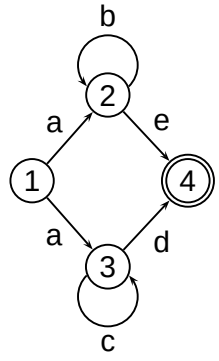
So, we create a new state Q representing that set and add it to our table.

State	Set	a	b	c	d	e
P	1	2,3				
Q	2,3					



State Q goes to 2 on b, 3 on c, and 4 on either d or e.
So, we create a new states representing them and add them to our table.

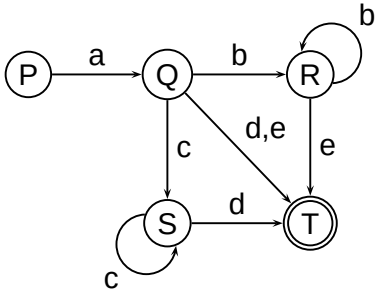
State	Set	a	b	c	d	e
P	1	2,3				
Q	2,3		2	3	4	4
R	2					
S	3					
T	4					



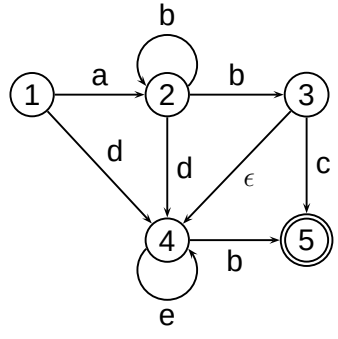
State R has inputs for b, and e. State S has inputs for c, and d. State T has no inputs.

State	Set	a	b	c	d	e
P	1	2,3				
Q	2,3		2	3	4	4
R	2		2			4
S	3			3	4	
T	4					

This gives us the instructions to make a new state machine that is deterministic.

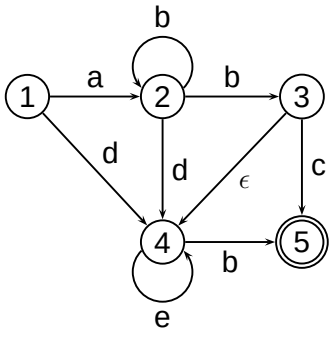


State	Set	a	b	c	d	e
P	1	2,3				
Q	2,3		2	3	4	4
R	2		2			4
S	3			3	4	
T	4					



State	Set	a	b	c	d	e
P	1	2			4	
Q	2					
R	4					

We add our initial state P, and see that we have inputs on a and d.

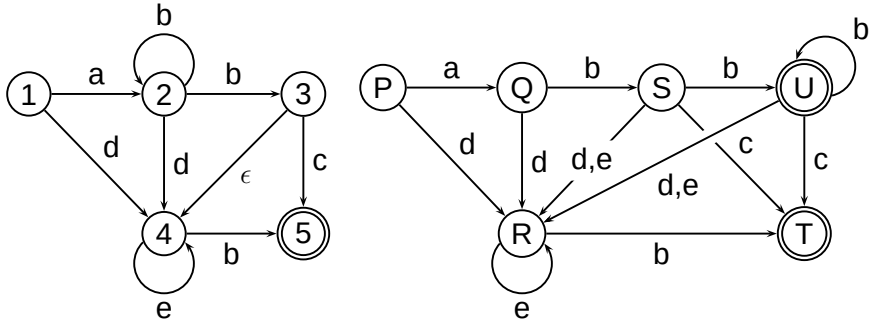


State	Set	a	b	c	d	e
P	1	2			4	
Q	2		2,3,4		4	
R	4		5			4
S	2,3,4					
T	5					

Note that from Q, input b gives us states $\{2, 3\}$, but we add $\{2, 3, 4\}$ to our table because of the ϵ-closure rule.

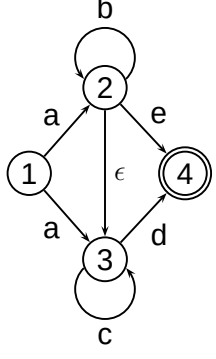
Finishing the table, we get...

State	Set	a	b	c	d	e
P	1	2			4	
Q	2		2,3,4		4	
R	4		5			4
S	2,3,4		2,3,4,5	5	4	4
T	5					
U	2,3,4,5		2,3,4,5	5	4	4

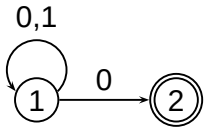


Note well: we have two accept states.

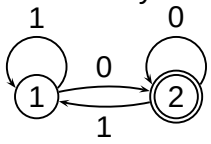
1. (129) Draw an NFA that accepts even binary numbers.
2. (130) Draw a DFA that accepts even binary numbers.
3. (131) Convert this example into a DFA.



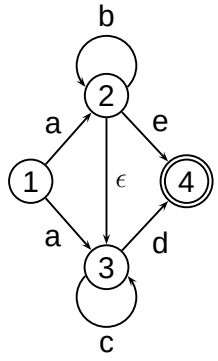
Draw an NFA that accepts even binary numbers.



Draw a DFA that accepts even binary numbers.

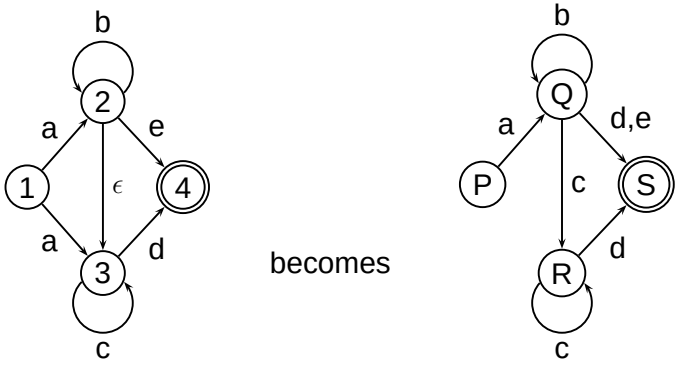


Convert this example into a DFA



State	Set	a	b	c	d	e
P	1	2,3				
Q	2,3		2,3	3	4	4
R	3			3	4	
S	4					

The DFA



Motivation

- *Regular Languages* were developed by Noam Chomsky in his quest to describe human languages.
- Computer Scientists like them because they are able to describe “words” or “tokens” very easily.

Examples:

Integers a bunch of digits

Reals an integer, a dot, and an integer

Past Tense English Verbs a bunch of letters ending with “ed”

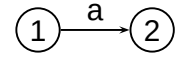
Proper Nouns a bunch of letters, the first of which must be capitalized

A bunch of digits?!

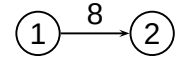
- We need something a bit more formal if we want to communicate properly.
- We will use a *pattern* (or a *regular expression*) to represent the kinds of words we want to describe.
- As it will turn out, these expressions will correspond to NFAs.
- Kinds of patterns we will use:
  - Single letters
  - Repetition
  - Grouping
  - Choices

Single Letters

- To match a single character, just write the character.
- To match the letter “a”...
  - Regular Expression: `a`
  - State machine:

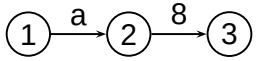


- To match the character “8”...
  - Regular Expression: `8`
  - State machine:

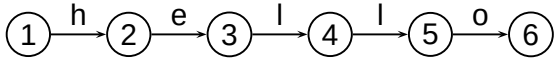


Juxtaposition

- To match longer things, just put two regular expressions together.
- To match the character “a” followed by the character “8”...
  - Regular expression: `a8`
  - State machine:



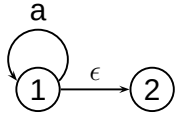
- To match the string “hello”...
  - Regular expression: `hello`
  - State machine:



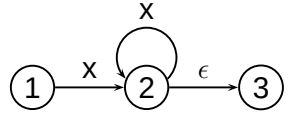
Repetition

- For zero or more occurrences, add a `*`
- For one or more occurances, add a `+`

- Zero or more copies of a...
  - Regular expression `a*`
  - State machine:



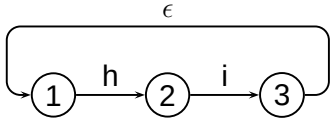
- One or more copies of x...
  - Regular expression `x+`
  - State machine:



- For those of you who have used unix or DOS filename matching, this should look familiar.

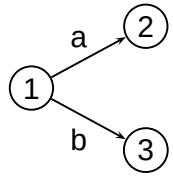
Grouping

- To groups things together, use parenthesis.
- To match one or more copies of the word “hi”...
  - Regular expression: `(hi)+`
  - State machine:



Choice

- To make a choice, use the vertical bar (also called “pipe”).
- To match an a or a b...
  - Regular expression: `a|b`
  - State machine:



Examples

Expression	**(Some) Matches**	**(Some) Rejects**
`ab*a`	`aa`, `aba`, `abbba`	`ba`, `aaba`, `abaa`
`(0|1)*`	any binary number, ϵ	
`(0|1)+`	any binary number	empty string
`(0|1)*0`	even binary numbers	
`(aa)*a`	odd number of as	
`(aa)*a(aa)*`	odd number of as	

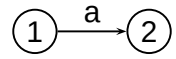
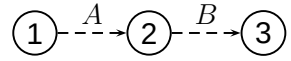
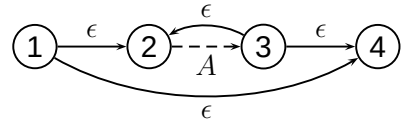
`(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`
even number of as and b

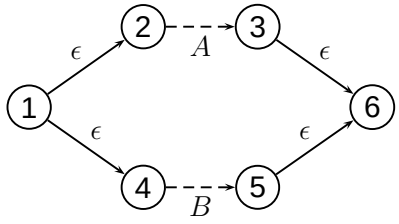
Some Notational Shortcuts

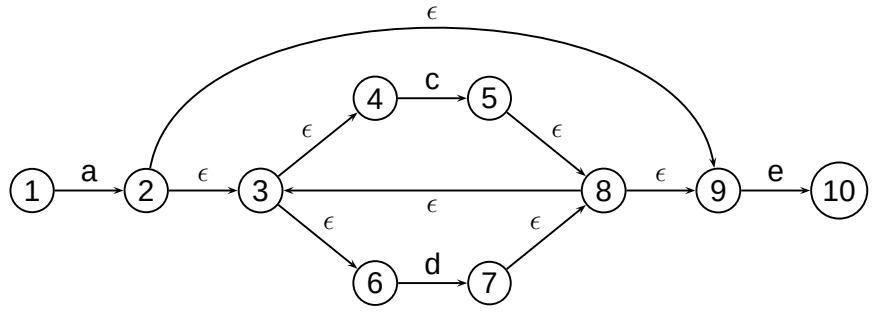
- A range of characters: `[Xa-z]` matches `X` and between `a` and `z` (inclusively).
- Any character at all: `.`
- Escape: `\`

Expression	**(Some) Matches**
`[0-9]+`	integers
`X.*Y`	anything at all between an `X` and a `Y`
`[0-9]*\.[0-9]*`	floating point numbers (positive, without exponent

- They are *greedy*.
  `X.*Y` will match `XabaaYaababY` entirely, not just `XabaaY`.
- They *cannot count* very well.
  - They can only count as high as you have states in the machine.
  - This regular expression matches some primes:
    `aa|aaa|aaaaa|aaaaaaa`
  - You cannot match an infinite number of primes.
  - You cannot match "nested comments". (`\*.*\*`)

Regular Expression	**State Machine**
Single letter a:	
Juxtiposition of A and B	
Repitition of A	

Regular Expression	**State Machine**
Choice between A and B	



These are highly unoptimized, but very easy to compose together. You can use the techniques from last lecture to optimize it.

Problems I

Write a regular expression for the following kinds of words

- hexadecimal numbers
- numbers in scientific notation
- file names ending in `.C`
- numbers between 0 and 255

Describe in English the following regular expressions

- `[a-zA-Z][a-zA-Z0-9]+`
- `[a-z]*(es|ed|ing)`
- `<[a-z0-9]+@[a-z0-9]+(\.[a-z0-9]+)+>`

Asnwers I

- hexadecimal numbers: `[0-9A-Fa-f]+`
- numbers in scientific notation:
  `[0-9]+\.[0-9]+E(+|-)[0-9]+`
- file names ending in `.C`: `.*\.C`
- numbers between 0 and 255:
  `25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9]`
- `[a-zA-Z][a-zA-Z0-9]+` like variable names
- `[a-z]*(es|ed|ing)` words ending in “es”, “ed”, or “ing” (verb forms)
- `<[a-z0-9]+@[a-z0-9]+(\.[a-z0-9]+)+>` email addresses

Problems II

Which of the following can be described by regular expressions?

- All the words in the English language
- All the Fibonacci numbers
- “All Your Base Are Belong To Us” video
- Numbers that are multiples of 4 (assume $>=$ 2 digits)
- Words that have exactly as many as as they have bs
- Pallindromes

Answers II

- All the words in the English language
  Yes — it’s huge, but it works. `(a|aardvark|abate|...`
- All the Fibonacci numbers
  No — the set is infinite and requires computation
- “All Your Base Are Belong To Us” video
  Yes — again, huge, but it works
- Numbers that are multiples of 4 (assume $>=$ 2 digits)
  Yes — `[0-9]*([02468][048]|[13579][26])`
- Words that have exactly as many as as they have bs
  No — requires unbounded counting
- Pallindromes
  No — requires unbounded memory
  (aibohphobia = fear of pallindromes)